

Dios Architecture

Updated on 01/13/2006

The Dios has 33 general purpose IO pins. Some of the IO ports also have other functionality as well. For instance ports 16 and 17 also double as the programming ports. Ports 8 and 9 can be used as a hardware serial port.

The Dios has a self contained 3rd generation command processor. The compiler features KRCompression technology.

The bulk of the command processor resides in its own 16k of memory space on the chip. User programs are stored in another 16k of space. With KRCompression the memory usage for commands are quite low. For instance I ported an 8k application from another microcontroller over to the Dios and it only used 2k of space. Hence 16k on the Dios is like having 32-64k on other microcontrollers.

The Dios has remarkable performance.

Here are a few of speed comparisons

Micro Controller	Single high/low command	multiple high/low commands
Dios	43,800 Hz	49,200 Hz
Atom	13,638 Hz	16,688 Hz
Pic16F628 (20mhz) w Mbasic compiler	10,232 Hz	11,834 Hz
Basic Stamp 2p	4,651 Hz	9,691 Hz
Basic Stamp sx	4,191 Hz	7,448 Hz
Basic Stamp 2	1,768 Hz	3,025 Hz
Athena / Athena485	5,000 Hz	6,280 Hz
Perseus	10,100 Hz	12,680 Hz
AthenaHS	25,400 Hz	31,740 Hz

All controllers were running the same program. As you can see the Dios blows them out the door. It does this with a special base set of tokens that sync well with the host controller architecture. The only thing faster is going to be raw assembly language (which the Dios can do as well). We have spent over 2 years optimizing and perfecting the Dios engine.

[Math](#)

[Memory Layout](#)

[Variables and Constants](#)

[Bit fiddling](#)

[String Variables](#)

[Functions](#)

[Arrays](#)

[IRQ's](#)

MATH

The Dios Supports two types of math. 16 bit integer math and 32 bit floating point math.

In many cases the two are interchangeable but there are some subtleties that will be noted at the end of this topic.

Integer Math

Integer math on the Dios is very simple. In most cases 16 bit math used except in the case of the multiplication the result can be up to 32 bits. All math is unsigned.

The following operations are supported

- * Multiplication
- / Division
- + Addition
- Subtraction
- & And
- | Or
- ^ Xor
- // Return the Remainder after a Division
- ** Return the high word after a multiplication

You can use math any where an expression is used or in variable assignments.

Example

```
dim res as integer
res = 10 + 5
```

Floating Point Math

Floating point math is very similar to integer math. However larger numbers and signs are supported along with fractions.

Floating point is built into the Dios engine and its use does not use any more memory than integer use.

- * Multiplication
- / Division
- + Addition
- Subtraction

You can use math any where an expression is used or in variable assignments.

Example

```
dim res as float
res = 10 + 5
```

Floating Point Exceptions and rules

Any expression that expects a integer will have each component converted to integer before insertion into the expression.

Examples

- Integer variables in assignments
- Most built in commands
- Functions arguments defined as integer

Any expression that expects a floating point value will have each component converted to 32 bit floating point before insertion into the expression

Examples

- Floating point variables
- Function arguments defined as float

All commands that require variables to hold target data are restricted to integer variables.

The following commands can be used with floating point variables.

exceptions

- while/wend
- if/then/else
- address
- clear

Conversion

Binary Conversion

you can assign a binary number to a variable or as part of an expression by preceding the number with a %.

Example

```
res = %100
res will contain 4. Note that the MSB is on the left side.
```

Hex Conversion

you can assign a hexadecimal number to a variable or as part of an expression by

preceding the number with a \$.

Example

```
res = $41  
res will contain 65.
```

ASCII conversion

You can convert an ASCII character to a byte by enclosing the character in ''

Example

```
res = 'A'  
res will contain 65
```

Memory Layout

- 256 bytes used for 16 bit integer local variables and 32 bit Floating Point variables available to each function (takes space off the stack)
- 256 bytes used for 16bit integer and 32 bit global global variables
- 256 bytes string space.
- 256 bytes of eeprom memory
- 256 byte serial buffer
- 16K of user program space.

All memory locations are readable and writable by the user.

Variables and Constants

There are three types of variables in the Dios. 16 bit integer variables, 32 bit floating point variables, and string variables. Both integer and floating point variables may be assigned as ether local or global variables. String variables can only be assigned as global.

Local Variables

When a variable is local the space it takes will be freed up once the function in which it was declared has exited.

Example

```
Func main()  
Dim myvarb1  
Displaystuff()  
Endfunc  
Func Displaystuff()  
Dim x,y,z  
print x," ",y," ",z  
Endfunc
```

The three variable slots taken up by the x,y,z variables in function Displaystuff will be freed up for use by other functions once the Displaystuff function has exited.

Why do this? The features above make for a lot more efficient use of memory. It makes it easier on the programmer to keep track of things and to modularize code.

To declare a local variable use the **Dim** statement.

To create a integer variable use:

```
dim xyz as integer  
or  
dim xyz
```

To create a floating point variable use:

```
dim xyz as float
```

Global Variables

When a variable has been defined as a global variable it can be accessed by all functions. Global variables never go away.

To declare a global variable use the statement **global**. Global variables may be declared any where. Even outside functions.

To create a global integer variable use:

```
global xyz as integer  
or  
global xyz
```

To create a global floating point variable use:

```
global xyz as float
```

Note if you wish to access the different components of a integer variable such as the individual bits and bytes check out the [Bit Fiddling help topic](#)**[Link=150]**.

Byte Variables

You can also access/modify a normal integer variable through its `.byte(x)` modifier.

For example

```
myvarb.byte(0) = myvarb.byte(0) + 1
```

Constants

Constants are fixed numbers that are referenced by a useable name.

For example

```
const CLK 4
```

Would allow us to use the word CLK to represent the number 4. That way you can have several references to the CLK pin. To change the pin from 4 to 5 means only having to change 1 line of code.

Once a constant has been defined the value cannot be changed. In other words you **can't** have the statement

```
CLK = 25
```

Constant statements are local to the functions that they were declared just like local variables.

Global Constants

You can also define global constants. That is constants that are available to all functions. To define a global constant use the **gconst** statement just like you would use the const as shown above.

Global constants can be defined any where in the program file.

Note: Constants may hold both floating point and integer values.

Variable scope

It is possible to have a global variable called test and a local variable called test. In this case the local variable will take precedence while in the function that created the local variable. The same is true for constants.

Bit Fiddling

Variables are made up from two memory locations (bytes). Each byte consists of 8 bits. You can access both the individual bytes and bits of a variable by using extensions.

Byte Extension

.byte(x)

To access each byte of a variable use the byte extension. Where x is the byte number. Use 0 for the low order byte and 1 for the high order byte.

Example 1

```
func main()
  dim a
  a = 1000
  print a.byte(0)
  print a.byte(1)
endfunc
```

This example will print out 232 and 3.

You can also set a variables individual bytes as in the following example.

Example 2

```
func main()
  dim a
  a.byte(0) = 232
  a.byte(1) = 3
  print a
endfunc
```

This example will print out 1000

Bit Extension

.bit(x)

To access each bit in a variable use the bit extension. Where x is the bit number 0-15.

Example 3

```
func main()
  dim a
  a = 3
  print a.bit(0)
endfunc
```

This example will print out 1

To set the bit is just as easy.

Example 4

```
func main()
  dim a
  a.bit(0)=1
  a.bit(1)=1
  print a
endif
```

This example will print 3.

Note that when setting the bit that if the expression = 0 then the bit will be set to 0 anything else will make it a 1

Register Access

Accessing hardware and software registers is identical to variables except they are all only single bytes so there is no byte extension.

Example 5

```
func main()
  output 0
loop:
  PORTB.bit(7) = 0
  PORTB.bit(7) = 1
  goto loop
endfunc
```

This example will toggle IO port 0 by directly accessing the hardware register.

String Variables

String variables allow you to manipulate text data. You can define a string variable by using the global command.

global myvarb(20) as string

This defines a 20 byte string variable. You can store up to 19 characters in this variable. The last position is reserved for use as a string terminator. There is only 256 bytes of string memory available so use it wisely. The use of the table command can aid you in cutting down string space usage.

Once a variable has been defined it can be assigned values.

Direct Assignment

```
myvarb = "jump"
```



```
myvarb will contain jump  
myvarb2 = myvarb+" down"  
myvarb2 will contain jump down  
myvarb = * + " up"  
myvarb will contain jump up
```

String Insertion

```
myvarb="ABCDEFGH"  
myvarb(3)="mike"  
myvarb will contain ABCmikeH  
myvarb(2)=90  
myvarb will contain ABZmikeH  
myvarb(5)="1234567"  
myvarb will contain ABZmi1234567
```

Partial string Assignment

```
myvarb = "ABCDEFGG"  
myvarb2 = myvarb(1,5) 'Starting character 1 for 5 characters  
myvarb2 will contain BCDEF  
myvarb2 = myvarb(1,200) 'Starting character 1 for 200 characters or end of string  
myvarb2 will contain BCDEFG
```

There are some restriction in using string variables usage.

You can not assign a string variable to its self. For instance `myvarb = myvarb + "s"` is not allowed. If you need to add something to the end of an existing string use the `*` operator. This tells the string pointer to move to the end of the string.

You can not pass a variable to a function or return a string from a function. You can however pass a reference to the string so that the function can compare or manipulate it.

The math operators do not work the same when using strings. For instance.

```
myvarb=65  
myvarb will contain A  
myvarb=90+89+87  
myvarb will contain ZYX
```

Assigning a integer variable to a string works the same way. The lower 8 bits are converted to ASCII and added to the string.

There is no bounds checking on string assignments. So if you assign something out side of a string defined size it will be inserted into the next string. This could allow you to index many strings with a bit of math.

Once a string is defined its contents are unknown until you assign something to it.

String Terminator

An end of a string indicated by a 0 value character. Hence a string "jump" will actually contain 106,117,109,112,0

Most of the manipulation of the string terminator is done automatically. You can also manipulate the terminator.

```
global myvarb(20) as string
myvarb=0
or
myvarb(0)
will create an empty string
myvarb="ABCDEFGF"
myvarb(3)=0
myvarb will contain ABC
```

Notes on string insertion

As long as the insertion is located inside the bounds of the original string no terminator will be inserted.

```
globalvarb1(10) as string
varb1="ABCDEFGH"
varb1(3)=90
varb1 will contain ABCZEF GH
```

If the insertion point starts inside the bounds and extends outside the bounds a new terminator will be added to adjust the length.

```
globalvarb1(20) as string
varb1="ABCDEFGH"
varb1(5)="1234567"
varb1 will contain ABCDE1234567
```

If the insertion point is outside the original string no terminator will be inserted.

```
global varb1(10) as string
globalvarb2(10) as string
varb1="ABCDEFGH"
varb2="123456789"
varb1(13) = "mike"
varb1 will contain ABCDEFGH
varb2 will contain 123mike89
```

String Address Operator

You can use the string address operator to assign a string based on an address. Let's take the following example.

```
global varb1(20) as string
global varb2(20) as string
note that the address of varb2 is 20 (end of varb1)
varb2="Mike"
varb1=@20
```

The address operator takes the integer expression value and treats it like a string address. This can be useful in quick string or table access with functions where the address is passed.

Arrays

You can create both floating point and integer arrays. An array is a block of the same kind of variable.

Integer arrays

```
dim myvarb(10) as integer
Will create a block of 10 16 bit integer variables. You can access each individual
element by using the index.
myvarb(0) = 5000 'The first element in the array
myvarb(9) = 400 'The last element in the array
```

Floating Point Arrays

Float arrays work just like integer arrays

```
dim myvarb(10) as float
```

This will create a block of 10 floating point variables. Again you can access each element by using an index.

```
myvarb(1)= 2.76 '2 nd element
myvarb(2)=300.54 '3 rd element
```

Byte Arrays

while there are only 16 bit integers you can access the individual bytes in a single byte array using the byte index.

```
dim myvarb(10) as integer
```

Creates a block of 10 16 bit integers or 20 8 bit integers. You can access each byte by using the .byte extension as in

```
myvarb.byte(0) = 17 'First byte element
```

```
myvarb.byte(6) = 32 '7th element
```

Array exceptions and rules

You can use arrays in any expression. Automatic conversion will take place between each type.

You can not pass the whole array to a function. Only a single element can be passed. You can use global variable arrays to manipulate array data in different functions

You can not return whole arrays with the exit command. You can only return a single element.

When a command requires a variable for data return you may not use an array.

You can not access individual bits and bytes of array elements. For example myvarb(1).bit(7) is not currently allowed..

There is no bounds checking on arrays so if index outside the allocated block you will access data in other variables. This can be used as a fast access or can cause problems.

```
dim a(5) as integer
```

```
dim b(5) as integer
```

```
a(5)=10 This will access the 1st element in the b variable.
```

Functions

Dios has the ability to create functions. Functions can have any number of parameters passed to them. Both floating point and integer values may be passed. To pass a floating point value you must declare the argument as a float parameter. IE. func myfunc(data1 as float,data2 as float, data3 as integer, data 4) data1 and data2 will expect a floating point value passed to them. If it is not it will be converted. data3 and data4 will expect an integer value passed to them. If not it will be converted.

Functions can also return a 16 bit integer or 32 bit floating point value.

All functions take space from the stack for there local variable so the space is freed once the function exits. All functions have access to global variables. This allows you to easily manage several hundred variables in very little space.

Functions are totally self-contained with their own goto's and gosub's and local constants.

This will allow me to place several libraries on the website that can be used with very little modification.

Example

```
func main()
dim a
loop:
a=get831(3,4,5)
print a
goto loop
endfunc
func get831(Dat,CLK,CS)
dim retdata
'Put the connected pins into correct mode
output CS,CLK
input Dat
'Select the chip and sets things in motion
low CS,CLK
pulseout CLK,1 'This starts the stream
shiftdat Dat,CLK,200,retdata '8+64+128
high CS
exit retdata
endfunc
```

The above program will return the result from a ADC831 Analog to digital chip. It is only about 100 bytes long.

Setting a functions return type

By default when a function returns a value it returns a 16 bit integer. To return a floating point value you need to define the function as a float value as shown here.

```
func myfunction() as float
.....
endfunc
```

Note: If a function is set to return a float and its used in a integer expression the floating point value will be converted to integer. This will allow you to write generic libraries.

Optional parameters

You can set up functions so that they have optional parameters.

This is done by reading the software register OPP8 which contains the number of parameters that were passed. Note that the OPP8 register must be access as the first command as other commands and function calls will change its value.

Remember floating point parameters use to stack slots.

Here is an example

```
func doit(port,timeout)
'OPP8 is a Register that contains count of variables passed.
if OPP8 < 2 then
timeout = 5000
endif
.
.
.
endfunc
```

Passing string data

When passing string, table or text data to a string the actual data is not passed. A reference to the data is passed. In order to access this data you must collect the data using the `getstringbyte` command.

`disptext("hello")`

The interpreter converts the string into three parameters that are read by the called function as shown below.

```
func disptext(addr)
dim char
again:
getstringbyte addr,char,done
print char
goto again
done:
endfunc
```

IRQ's

IRQ's (interrupt requests) are a double edged sword.

Advantages

They can give us the ability to collect data in the background but they can also cause us a great deal of difficulty. For instance I can add a simple 32Khz clock crystal to the Dios and with interrupts create a very accurate real time clock.

They can wake the Dios up from sleep mode

Disadvantages

The disadvantage to this simplicity is that the internal timings of various asynchronous commands will be affected.

For instance the serous and debug commands may contain glitches. The pulsein and count commands will not yield accurate results.

You can temporarily suspend interrupts by issuing: `INTCON.bit(7)=0` This will stop all IRQs from firing. After your code has done what it needs you can then turn them back on again with the `INTCON.bit(7)=1` command. Just keep in mind that while the interrupt is off then you may miss an event.

How does the Dios handle IRQ's

There are two levels of complexity in dealing with IRQ's. The hardware level (18F52 chip) and the software level (Dios).

When a hardware IRQ is setup it will fire in the background regardless of what the Dios is doing. If you have flagged a hardware IRQ to call an `onirq` function this is a software IRQ. Software IRQ's are called at command intervals. If you issue a command that takes a long time you can create a lag situation.

Hardware level

In order to use IRQ's you must have a working knowledge of the hardware registers used in the (18F52) or at the very least access to the data sheet.

These are the hardware register associated with interrupts.

INTCON
INTCON2
INTCON3
PIR
PIR2
PIE
PIE2

Global Interrupt Enable

In order for any interrupt to work the global interrupt flag must be set. To enable this interrupt we use the `INTCON.bit(7)=1` command.

Peripheral Interrupt Enable

Some Hardware features inside the (18F52) chips will require that you enable the peripheral interrupts as well. Again this is done with the `INTCON.bit(6)=1` command.

Individual Interrupt Enable

You must also enable an individual interrupt as well. For instance we will enable the `INT0` interrupt. This interrupt is tied to `B0` (Dios IO Port 7). To enable it just issue the following commands.

```
INTCON.bit(7) = 1 'Turn on global flag
INTCON.bit(4) = 1 'Turn on INT0 interrupt
```

Once turned on every time there is a change in state the `INT0` interrupt will fire. You can even change which state change causes the interrupt. High to Low or Low to High.

Once an interrupt fires there is not allot that can be done on the hardware side. This is where the Dios steps in.

Software Level

Here are the things the Dios can do when a hardware interrupts fires.

- Set a flag
- Reset The interrupt
- Turn off the interrupt
- run a Dios Function
- run an irq assembly handler

Each interrupt has a status byte called `IRQxxx` where `xxx` is one of the following IRQ names.

Valid IRQ names

- `INT0` Interrupt on state change `B0` (Dios IO port 7)
- `INT1` Interrupt on state change `B1` (Dios IO port 6)
- `INT2` Interrupt on state change `B2` (Dios IO port 5)
- `TMR0` 8 or 16 bit timer/counter
- `TMR1` 8 or 16 bit timer/counter

TMR2 8 bit timer/counter
TMR3 8 or 16 bit timer/counter
CCP1 Capture/Compare or PWM output (Dios IO port 13)
CCP2 Capture/Compare or PWM output (Dios IO port 4)
AD Analog to digital conversion complete
RB State change on RB or (IO port 0-7)
TX Uart transmit buffer empty
LVD Detected low voltage threshold on power pins
SSP Internal SPI and I2c hardware port
PSP Internal Parallel port

Various bits in the IRQ flags are used for reading status and setting operations.

Status Register bit names

IRQFLAG
ONESHOT
ONIRQFLAG1
ONIRQFLAG2
ONIRQFLAG3

IRQFLAG

If set to 1 indicates that at least 1 IRQ has fired. Internal hardware IRQ's can fire many times before we can check them. This flag indicates that the IRQ has fired at least once. It is up to the software to clear this flag.

ONESHOT

If this bit is set then the hardware IRQ will fire only once. Use this to detect an event. Use the IRQFLAG to poll for a change.

ONIRQFLAG1
ONIRQFLAG2
ONIRQFLAG3

These flags are used internally by the **onirq** and **exitirq** commands

Jumping to a Dios function when a IRQ fires.

You may set up a function to be called when an IRQ fires. These functions are asynchronous to the hardware IRQ's That is they do not fire one for one with a hardware flag. Here is what happens.

1. hardware IRQ fires.

The hardware IRQ handler sets a flag that a SW function is to be called before the next command fetch. The IRQ will continue to fire based on the operation bits in the IRQ status flag.

2. Just before the next command fetch the flags are checked to determine if a `onirq` call was flagged. If it was then a call to the function is made. Note that no other calls to this function are going to happen while we are waiting to call this function or inside this function. The `onirq` is turned off.

3. If we exit the `irq` function normally with the `endirq` command no other call to this function will occur. If we exit with a `exitirq` command then the `onirq` is turned back on for this function.

It is up to the called function to utilize the various flags and counters to determine how many actual interrupts have occurred.

IRQ functions are set up with the `irqfunc` command and terminated with the `endirq` command. They can not be called as normal functions and only the `exitirq` command is a valid alternative to exit an IRQ function.

Example

Here is a small example that shows how to create `onirq` call based on IO port changing.

```
func main()
output 2
INTCON.bit(7)=1
INTCON.bit(4)=1
onirq INTO,procpulse
loop:
toggle 2
goto loop
endfunc
```

```
irqfunc procpulse()
dim x
print "State Change"
exitirq INTO
endirq
```

Jumping to a irq assembly handler when a IRQ fires.

You may set up an assembly handler when an IRQ fires. These routines are synchronous to the hardware IRQ's That is they fire one for one with a hardware flag. Here is what happens.

1. hardware IRQ fires.

The IRQ will continue to fire based on the operation bits in the IRQ status flag.

2. If a `startirqasm` routine has been setup for the fired interrupt it will be called.

Note that Both assembly handler routines may be called on the same routine. The assembly handler is always called first

IRQ assembly routines are set up with the **`startirqasm`** command and terminated with the **`endirqasm`** command. They can not be called as normal functions.

Example

Here is a small example that shows how to create `onirq` call based on IO port changing.

```
func main()
output 2
INTCON.bit(7)=1
INTCON.bit(4)=1
onirq INTO,procpulse
loop:
toggle 2
goto loop
endfunc
```

```
irqfunc procpulse()
dim x
print "State Change"
exitirq INTO
endirq
```